Artificial Intelligence for Engineering Design, Analysis and Manufacturing (1998), *12*, 307–320. Printed in the USA.
Copyright © 1998 Cambridge University Press 0890-0604/98 $12.50

Generative constraint-based configuration of large technical systems

MARKUS STUMPTNER,[1] GERHARD E. FRIEDRICH,[2] AND ALOIS HASELBÖCK[3]

[1] Technische Universität Wien, Institut für Informationssysteme, Paniglgasse 16, A-1040 Wien, Austria
[2]Universität Klagenfurt, Institut für Informationstechnologie, Universitätsstraße 65-67, A-9020 Klagenfurt, Austria
[3]Siemens AG Österreich, PSE EZE, Erdberger Lände 26, A-1030 Wien, Austria

(RECEIVED November 1, 1997; ACCEPTED February 5, 1998)

Abstract

This paper describes the technical principles and representation behind the constraint-based, automated configurator COCOS. Traditionally, representation methods for technical configuration have focused either on reasoning about structure of systems or quantity of components, which is not satisfactory in many target areas that need both. Starting from general requirements on configuration systems, we have developed an extension of the standard CSP model. The constraint-based approach allows a simple system architecture, and a declarative description of the different types of configuration knowledge. Knowledge bases are described in terms of a component-centered knowledge base written in an object-oriented representation language with semantics directly based on the underlying constraint model. The approach combines a simple, declarative representation with the ability to configure large-scale systems and is in use for actual production applications.

Keywords: Configuration; Generative Constraint Satisfaction

1. INTRODUCTION

Over the years, configuration systems for technical domains have been among the most industrial applications of artificial intelligence technology, dating back to the R1/XCON system (McDermott, 1982), and many other systems described in the literature (e.g., Bowen, 1985; Searls & Norton, 1990; Yokoyama, 1990). Reasons for this success story include the availability of well-structured, limited domains, where in addition at least part of the knowledge is available in written, coherent form (i.e., technical documentation), and fairly clear specifications for checking validity and evaluating quality of solutions exist.

Many of the early knowledge-based configurators, however, were exclusively rule-based, closely coupling domain and strategic knowledge, leading to problems with the design and maintenance of large knowledge bases (Barker & O'Connor, 1989). This, in addition to the desire to better describe the fundamental properties of the configuration domain (or, as it is sometimes called, configuration design), led to a variety of approaches that improved on the original approach.

Two central steps in this process were the proposal of a more rigorous model for the representation of configuration knowledge by Mittal and Frayman (1989), expressed in terms of components and ports, with reasoning starting out from certain so-called *key components* and consisting of the incremental addition of required auxiliary components and any new connections (ports) associated with them. They chose to encode this situation as a constraint satisfaction problem (CSP) (Frayman & Mittal, 1987). A CSP is defined by a set of variables V, where each variable $v \in V$ has an associated domain $dom(v)$ that designates all values this variable can possibly assume. Constraints are relations defined on sets of variables that specify what *combinations* of values may be assigned. Formulating configuration and design requirements as constraints is a quite natural choice, thus it is not surprising that other projects have taken this approach (Frayman & Mittal, 1987; Steinberg et al., 1989; Cunis et al., 1989; Birmingham et al., 1989).

The intent behind the research and system described in this paper was to design a tool for technical configuration,

Reprint requests to: Markus Stumptner, Technische Universität Wien, Institut für Informationssysteme, Paniglgasse 16, A-1040 Wien, Austria. E-mail: mst@dbai.tuwien.ac.at.

https://doi.org/10.1017/S0890060498124046 Published online by Cambridge University Press

based on a limited set of core concepts, with a declarative form of representation, but the ability to configure large and very large systems. Starting from the classical CSP configuration point of view, a formalism was developed that could handle differing quantities of components while maintaining adequacy in structural descriptions. In Section 2, we give a description of an example application domain and characterize the dimensions along which knowledge for configuration purposes can be classified. In Section 3, we then introduce the constraint-based representation language LCON used in the COCOS project, and the underlying GCSP (Generative CSP) formalism in Section 4. In the final sections of the paper, we discuss the COCOS system architecture and the experience gained from actual industrial use of the tool.

2. REPRESENTING CONFIGURATION KNOWLEDGE

Configuration is generally considered a subset of the field of design. Whereas design concerns itself with the conception or construction of new artifacts in the most general sense, configuration is usually taken to deal with the composition of artifacts from a limited set of smaller components. While architectural examples are sometimes described (configuring the floor spaces of an office building depending on the offices, passages, lifts and stairs and maintenance spaces required), most configuration research deals with technical systems. The XCON system, the classic case of a configuration system, configured computer systems, the VT system (Marcus et al., 1988), which has reached a certain fame in the knowledge acquisition community (Schreiber & Birmingham, 1996), produced elevator designs, and other systems which configure computers and telecommunications equipment.

As defined in Brown and Chandrasekaran (1989), design is an activity that produces an artifact from a set of primitive components given a repertoire of possible primitive relations between the components, noting that in general in realistic domains the set of primitive objects will be quite large and not explicit, at least initially. The main restriction which unites the examples of configuration tasks in the previous paragraph is that their set of primitive objects is generally explicit:

- For the floor space assignment problem, the objects could be the different types of rooms (office, lift, stairwell, toilets, maintenance spaces, air ducts, etc.).
- For the XCON system, they are computer housings, different kinds of circuit boards from CPU boards to controllers, power supply units, tape or disk drives, etc.
- For the VT system, they included the elevator cabin, motors, cables, and structural members.
- For telecommunications equipment, they include different kinds of functional components (communications equipment, power supply, interface boards), larger components for housing them, cables and plugs, and others.

The set of relations is likewise restricted.

- For an architectural example, there would physical adjacency, the need for multiple rooms not to intersect on the same spot, and the existence of doors or passages.
- For the VT system, there were fixed connections for bearing the weight of the cabin and applying force to the cable for lifting the cabin (in fact, these were so fixed that they were not all made explicit in the system's representation).
- For computers and other electronic equipment, there would be physical connections (power and data cables with plugs, bus slots, …), and functional requirements (e.g., performance figures, memory requirements, number of terminal or network connections for a computer).

As a running example in this paper from here on, we use a telephone exchange, the digital telephone switching system EWSD produced by Siemens, as it exhibits most of the properties which were and are driving our research.

2.1. An example system

A complete EWSD system belongs to the most complex electronic systems being commercially produced today. Basically, its purpose is to take calls incoming in any of the thousands of lines connected to such a system and route them appropriately to another, outgoing line. The physical structure of the core of the system, the so-called Switching Network (SN), consists of a three-tier parts hierarchy. At the top level, the system consists of a number of racks, which in turn contain so-called frames. Frames contain the basic components of the system, which are called modules. Depending on its type, a frame may contain up to 32 modules, and a complete system with multiple racks may contain thousands of modules overall. The number of modules (and thus indirectly the number of frames, and racks) is determined by the performance characteristics specified by the customer. These may include the number of lines to be handled by the system, and also switching time, costs, etc. Figure 1 shows the overall structure.

Superimposed on the physical aggregate hierarchy is a functional hierarchy. Frames are logically grouped into *switching units*, but the frames of a specific switching unit need not be physically adjacent, nor is a switching unit restricted to a single rack. On the other hand, a rack may contain the frames of several switching units. An example of such functional units is shown in Figure 2 where two switching units, SU-1 and SU-2, are spread over three racks.

The goal is to compose a system that fulfills certain functional requirements. This can be achieved by creating an adequate set of switching units that are composed from basic switching modules and appropriate support modules, for

https://doi.org/10.1017/S0890060498124046 Published online by Cambridge University Press

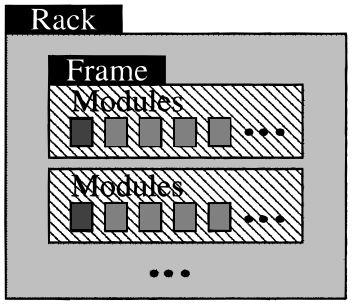


Fig. 1. Physical structure.

example, for power supply and testing. Constraints restrict the ways in which different components may be connected. Typical examples are:

- The frames in a given switching unit must contain either analog or digital modules only.
- The first slot of every frame is reserved for power-supply modules.
- The amount of electrical power consumed by all power-consuming modules in a frame has to be less or equal to the power output of the corresponding power-supply module.

Similar examples will be used for illustration in later sections of this paper.

2.2. Component-based configuration

The notion that is central to the configuration of technical systems is that of the *component*. The original notion of a component is that of a physical artifact. Components have properties whose values may be subject to variation (we will call them *attributes)*, and components can be connected to other components. The different types of connections then take the role of the relations in the previous section. If the complete system satisfies the original requirements, then the solution to the configuration problem is described by the set of components, the description of their attributes (numerical or discrete values), and the way in which they are connected.

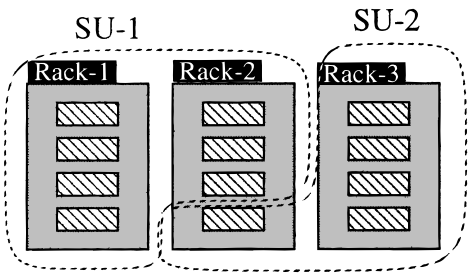


Fig. 2. Functional structure.

The *configuration process* then consists of the selection of an appropriate set of components from what is often called a *catalog* describing the domain, selecting their types, setting attributes, and choosing how to connect them.

It was decided at the outset to adopt a constraint-based representation for the COCOS system, for several reasons. Constraints are a natural way to express all kinds of compatibility knowledge, but there are additional advantages.

1. Constraints allow clear separation of problem and strategic knowledge, since strategies can be expressed declaratively, in terms of variable and value orderings (e.g., fill a frame starting with the leftmost slot) without changes to the constraints themselves.
2. A constraint-based representation can be used for either synthesis or verification: Given a finished configuration in terms of the constraint network that represents it, it is possible to check all constraints for inconsistency and thus test the validity of the configuration.
3. The simplicity and declarative nature of the basic constraint model allows the definition of clear semantics.

For configuration problems, constraint satisfaction (CS) techniques were embraced in a number of prototype systems (Mittal & Araya, 1986; Cunis et al., 1989) and form the foundation of the configuration approach described in Mittal and Frayman (1989), which focuses on the assembly of systems by connecting components.

EXAMPLE (from Mittal & Frayman, 1989). Assume a computer system has to be configured to meet an order specification of a customer. One of the required features is a printing function. The *key component* implementing such a printing function is a printer. After the selection of an appropriate printer type, additional components like interface cable or driver software must be chosen appropriately. This example demonstrates both function-component relationships and the idea of key components which, if chosen for configuration, result in the selection of a set of subordinate components. ■

However, the traditional constraint approach is limited in expressiveness and does not provide hooks for representing important features of configuration problems, namely, the organization in terms of components and the dynamicity of the problem, that is, the fact that the number of components may not be predetermined. In contrast to rule-based systems, where new components are explicitly introduced into working memory by actions specified in rule RHSs, the traditional constraint approach, while cleaner and more declarative, is effectively static. Mittal and Falkenhainer (1990) proposed *dynamic CSPs* (DCSPs) to avoid the latter problem, pointing out improved clarity and optimization possibilities.

EXAMPLE (from Mittal & Falkenhainer, 1990). The variable *package* with the domain {*luxury*, *deluxe*, *standard*} denotes the three offered car models in a car configuration

https://doi.org/10.1017/S0890060498124046 Published online by Cambridge University Press

domain. The *luxury* package provides an *air_conditioner*. In the notion of DCSP, this can be specified by the following constraint:

$$package = luxury \overset{\text{RV}}{\Longrightarrow} air_conditioner.$$

This constraint establishes the activation (i.e., inclusion of the corresponding component) of the *air_conditioner* variable in certain situations (if the *package* is *luxury)*. In addition, compatibility constraints restrict the configuration space. The following example states that, if both an *air_conditioner* of type *ac1* and an automatic sunroof opener are chosen, a battery of capacity *medium* must be included in the configuration:

$$air_conditioner = ac1 \ \& \ opener = auto \rightarrow battery = medium. \quad \blacksquare$$

However, the DCSP approach still suffers from strong limitations, since the component structure demanded in Mittal and Frayman (1989) remains unclear, while multiple occurrences of components with equal behavior are not supported at all. For example, in describing a twelve-cylinder engine, every constraint on cylinder behavior would have to be written down twelve times. Bowen and Bahler (1991) use a language based on the semantics of free logic to address this problem.

A different trend led to the development of strongly dynamics-oriented representation schemes for configuration. These were based on the realization that a central activity during configuration is the accumulation of entities that together provide for some needed functionality. The main assumption of the so-called *resource* paradigm is that functionality-based selection from a component library should be the central operation in a configuration system (Heinrich & Jüngst, 1991).

EXAMPLE. Consider another example from a computer configuration domain. The customer requests a PC with 64 MB RAM. The company has memory units of 16 MB capacity a piece in stock. A motherboard is initially equipped with 16 MB, but it has vacant slots for up to 256 MB. Thus, after the selection of the motherboard alone the working-memory resource shows a deficit. In the resource-based approach, configuration is fundamentally considered a task of resource balancing. In our example, the selection of an appropriate set of additional memory units will lead eventually to a compensation of the customer's memory requirements by the total amount of RAM provided on the motherboard. ■

The way in which these mechanisms interact with the constraint-based and connection-oriented view of configuration is handled in different ways by different systems. For example, Heinrich and Jüngst (1991) use rules and constraints for various parts of the knowledge base, integrated by a global blackboard and agenda. Our aim is to provide a formalism that expresses resource relationships clearly in CSP terms, while still providing a component-oriented view and the possibility of dealing with systems containing a multitude of similar, but perhaps differently, connected components.

The above discussion of configuration process implies a number of criteria by which suitable representations can be classified. Starting out from the CSP model, we will now explore these dimensions.

2.3. Strategy

Strategic knowledge concerns itself with the ordering in which components should be created, connections closed, and values chosen for properties. Its aim is to provide efficient reasoning and good-quality solutions. It is therefore usually heuristic in nature.

Heuristics can either be used to locally prefer one action over another, for example, during depth-first search, or by using a heuristic function to estimate solution quality, and then continue the search starting from the overall best-valued partial configuration.

Due to the huge search space, it was decided to avoid global optimizations and experiment with them later, if necessary. Local ordering heuristics and domain-independent constraint heuristics are used. It should be noted that this is a commonly acknowledged restriction. We do not know of any configurator in the literature that provides global optimization capability as well as the ability to tackle problems of comparable size.

2.4. Functionality

Functionality specifies why a certain type of component can be chosen for a solution. Basically, three kinds of functionality can be differentiated: *abstract functionality*, that is, the selected component needs to possess a certain named property at all (such as a printer's ability to print); *value-oriented functionality*, that is, the selected component needs to provide a certain value range for a given component attribute (e.g., power output); and finally, *structural functionality*, that is, the ability to establish a connection with another component *via* a so-called *port*. Note that this need not be a physical feature—for instance, need for a sort routine in a program would constitute a port on which to connect a suitable library during linking. One of the aspects which distinguishes configuration problems from design problems in general is that explicit function representations are not necessarily needed since most components in the catalog will already have been designed with a particular function in mind.

In the COCOS system, both attributes and ports are represented as CSP variables.

2.5. Quantity

This criterion represents the issue of how many components are needed, or, on a component-by-component basis,

https://doi.org/10.1017/S0890060498124046 Published online by Cambridge University Press

for what reason a new one must be introduced into the solution. Basically, the number could be fixed beforehand, it could be restricted by some upper limit, or it could be unknown before the final check. Traditionally, most configuration approaches represent the first kind of situation, where each component needed is known from the outset, just the exact type and parameter values are not. This is the view that lends itself most readily to an implementation as a CSP: For each component, there is one variable, and selecting the correct value for the variable means selecting the correct type of component. This approach, however, cannot deal with component quantities that are not predetermined.

Instead, component quantities can be made variable in two ways. The first is simply to record the number of components that occur for each type. If a quantitative relationship between different types can be established, solving the numerical equations and in equations posed by these relationships will result in a solution. For example, divide the number of modules by 32 to arrive at the number of frames, and choose enough frames to accommodate all modules. This is similar to the concept of a resource mentioned in Section 2.2.

However, since components (such as the modules and frames of the EWSD example) have individual properties (attributes or connections), each component needs to be represented as a separate object, and the appropriate parameters and ports must exist for each component separately. The conclusion is that if the quantity of components is not known beforehand, the creation of a component of a certain type will result in the creation of the appropriate set of attribute and port variables. The mechanisms used for this will be discussed in Sections 3 and 4.

2.6. Types and abstraction

Individual existence of components implies the presence of type entities independent of the existence of any particular component. If explicit types exist, one can try to group them into an inheritance hierarchy. Abstracting common features of two types A and B out into a common supertype C means that these features are now described without redundancy. On the individual component level, type abstractions also allow efficient reasoning. If at a given point in time it is not clear whether type A or B will be advantageous for a certain component, then the common properties and constraints represented by C can already be used to prune the search space, without precluding any choice (Mittal & Frayman, 1987).

While other configurators have used complex terminological reasoning mechanisms successfully (Wright et al., 1993), in COCOS, the decision was made to focus on instance-level reasoning and to use the simplest case for inheritance: have inheritance relationships specified by the knowledge base designer, disallow multiple inheritance, and define inheritance in a monotonic manner. That is, the attributes, ports, and constraints of the subtype are syntactically a superset of those of the supertype. This decision was taken since configuration domains typically encompass a fixed set of types, where technical documentation will usually support a taxonomic structure (if it does not already provide a ready-made one). Any component in a finished configuration must possess a specific type from that set. Therefore the situation (as it appears in classification systems) of generating a new "intermediate" class that exactly fits the available requirements for a given component will not occur. Monotonicity was decided upon since nonmonotonic inheritance mechanisms are computationally and conceptually complex, and the effectiveness and simplicity advantage gained from abstraction would be partially lost.

We will return to the computational aspects of the inheritance mechanism in Section 5.4.

2.7. Structure

This criterion deals with the kind of connections that can be generated. For example, can the graph of connections have arbitrary structure? Is it possible to distinguish between mandatory and optional connections (i.e., can a certain slot remain empty)? Can part-of hierarchies be represented? If yes, is this hierarchy fixed? All questions except the last can be answered "yes" for LCON. Note that Figure 2 shows two separate, but intermingled part-of hierarchies; one based on physical containment, and one on functional decomposition. Defining a single fixed part-of hierarchy would not be adequate in this situation. We will return to this issue in Section 5.3.

2.8. Compatibility

Compatibility knowledge expresses the validity of connections between certain components and the legality of attribute value combinations. This is the kind of knowledge that is represented in COCOS through constraints on port and attribute assignments.

3. EXPRESSING CONFIGURATION KNOWLEDGE IN LCON

LCON is the constraint language developed for the representation of configuration problems in the COCOS system. The basic concepts of LCON are similar to those of most ordinary, object-oriented KR systems, since the analogy between components (with identity, type, and descriptive properties) and objects is a very close one (Yokoyama, 1990). The main novel aspects of LCON lie in the fact that its semantics, despite being constraint-based and dealing with individual ports, are based on the generative constraint satisfaction (GCSP) formalism which allows reasoning about component existence. The GCSP semantics will be described in the next section.

The syntax of LCON was defined so the constraints could be written in a manner close to standard logical syntax.

https://doi.org/10.1017/S0890060498124046 Published online by Cambridge University Press

3.1. Component descriptions

A component type definition lists the ports and attributes of the component type. The following definition describes a frame (taken from the domain described in Section 2.1), which has the attributes `costs`, `name`, and `versions`, and the ports `mountedOn` and `slot1` up to `slot32` (as in Prolog, LCON constants have a lowercase first letter, and variables have an uppercase first letter). The set of ports is always extensionally specified, and finite. The term `frame F` signifies that the definition holds for all components `F` of type `frame`.

```
frame F DESCRIPTION:
 F.costs := 1234.5;
 F.name := 'F:DLU(A)';
 F.softwareVersions := {1.0, 2.0, 3.2};
 F.PORTS := {mountedOn[rack],
         slot1[powerSupply], slot2[module],
         ..., slot32[module]};;
```

Port `slot1` may be connected to components of type `powerSupply`, and the other slots may be connected to components of type `module`. Note that in no case does a port specify a connection to a specific component instance.

3.2. Inheritance

The component types in LCON are arranged in an inheritance hierarchy, by specifying ISA relationships between them:

```
subType ISA superType
```

As noted in Section 2.6, inheritance is monotonic, and multiple inheritance is disallowed. The root of the inheritance hierarchy is the type `component`. The leaves of the tree are the concrete component types available in the domain. A finished configuration will always contain only components which possess one of the leaf types.

3.3. Constraints

An LCON constraint is a statement that restricts the values of attributes and ports, either absolutely or in terms of their relationships to other attributes and ports (the latter may belong to other components). Constraints allow the formulation of complex design requirements, and the structure of the components describes the scope of constraints.

A constraint consists of a number of predicates combined by logical connectives (&, ∨, ->, NOT). Predicates for attributes and ports include equality and inequality. Ports can also check whether and to what component they are connected. The predicate `is_type` can be used to check whether a component possesses a particular type. Constraints are always specified in the context of a particular component type, which means only port functions and port sets of that particular component type may be used in their definition. Attributes and ports may be compared across types. Constraints can also be named and invoked as user-defined predicates in any constraint defined for the same type or a subtype.

3.3.1. Connections

Apart from the equality/inequality operators, the most important predefined predicate in LCON is `CONN`, which is used to specify connections. A literal of the form `CONN(C1,P1,C2,P2)` is true if the port of component `C1` specified by `P1` is connected to the port `P2` of the component `C2`. For example, the expression `CONN(F,slot15,M,mountedOn)` requires that the port `slot15` of component `F` be connected to port `mountedOn` of component `M`. In this case, `P1` (i.e., `slot15`) is a constant representing a port name. It could also be a port function (an extensionally defined function that maps a port to another port), as in `CONN(F,F.right_neighbor(slot14),M,mountedOn)`, or a port variable (discussed in the next section).

Note that connections specified by `CONN` are always bidirectional. If, for a given constraint, the identity of the connecting port or component is not relevant, this can be expressed through the special variable `_`, as in `CONN(Frame,right,M,_)`. This is true if the right port of `frame` is connected to some port of `M`. The following constraint definition

```
frame F DESCRIPTION:
 [ CONN(F,slot15,M15,_)&CONN(F,slot16,M16,_)
    ->((M15 = M16) or is_type(M15,moduleA)];
```

means that if both `slot15` and `slot16` are occupied, either the same component must be connected to both of them, or it must be of type `moduleA`.

3.3.2. Quantifying over ports

For convenience, ports can be grouped into named *port sets*. For example, slots can be grouped according to their placement in a frame (see Figure 3):

```
frame F DESCRIPTION:
 F.upperUnit := {slot1,...,slot16};
 F.lowerUnit := {slot17,...,slot32};
```

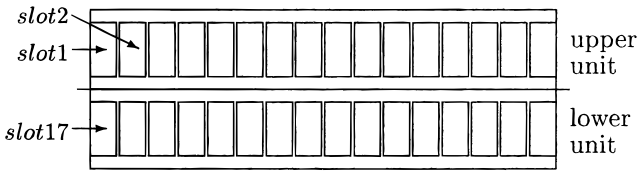


Fig. 3. A frame.

https://doi.org/10.1017/S0890060498124046 Published online by Cambridge University Press

The main use of port sets is as domains for port variables. A port variable can be universally or existentially quantified, and always ranges over a given port set. This allows a succinct description of the internal structure of large components. For example, the following constraint requires that no switching module (represented by the variable `Module`) connected to a port in the port set `upperUnit` may have a value for attribute `erlang` of more that 0.05.

```
frame F DESCRIPTION:
 ALL P IN F.upperUnit:
  [ CONN(F,P,Module,_) &
    Module is_type SwitchingModule
    -> NOT (Module.erlang >  0.05) ];;
```

Note that this constraint is bidirectional in the sense that if we assume the existence of a module in a slot of `upperUnit`, either its `erlang` value would be constrained to be smaller than 0.05, or its type would be constrained to be different from `SwitchingModule`.

3.4. Resource constraints

Certain requirements cannot be described by component-specific constraints, and instead must be described in more global terms. One such case is customer requirements, which will often not be given in terms of components, but in terms of overall functionality, for example, "the analog modules built into the finished system must have a total communications throughput of at least 5000 erlang units." Therefore, a special kind of constraints was introduced to deal with global parameters of a configuration. Since their semantics are similar to the resource concept presented in Heinrich and Jüngst (1991) which has gained wide acceptance in the configuration field, they are called *resource constraints*.

A resource constraint is an equation over several numeric expressions, which are called *resource functions*. Resource functions express the way in which attribute values (or component numbers) can be combined to express some kind of aggregate functionality. For example, the constraint from the previous paragraph would be written as

```
SUM{M.erlang | M ISA moduleAnalog} >= 5000;
```

The aggregate functions currently supported are `SUM` and `CARD`. The relations (comparison operators) specified in resource constraints must be linear. The following example specifies that power requirements must always be guaranteed:

```
SUM{M.powerOutp | M ISA component} >
 SUM{M.powerReq | M ISA component};
```

Resource constraints are one important source for the creation of components. We will discuss this topic in more detail in Section 5.

LCON provides a number of additional features, such as definition of certain constraints in tabular form, assignment of priorities to constraints (*soft constraints)*, and computed numerical attributes. We do not discuss them further in this paper.

4. GCSP AND THE SEMANTICS OF LCON

The semantics of an LCON program are defined by mapping it to an extended constraint satisfaction problem (called *generative* CSP, or GCSP). The designation *generative* stems from the fact that GCSP semantics require the generation of an adequate set of components as part of the solution process. An original basis of the GCSP formalism was presented in Stumptner and Haselböck (1993). We bring here a short overview of its current form.

A configuration problem to be solved by COCOS is specified by listing a set of initial variables and constraints on them. A solution is obtained as soon as all variables existing at that point, including all the initial variables, have been assigned a value (a concrete type in the case of component variables) and all constraints restricting these variables are satisfied.

DEFINITION 1 (Configuration domain). A *configuration domain* $\mathcal{D}$ *is a tuple* $\langle \mathcal{C}, P, \Gamma \rangle$, *where*

- $\mathcal{C}$ is a possibly infinite set of component variables. The domain of each variable from $\mathcal{C}$ is a finite set $T = \{\tau_1, \ldots, \tau_t\}$ of component types.
- $P = \{p_1, \ldots, p_m\}$ is a set of property names. Furthermore, there exists a set $D = \{\mathcal{P}, D_1, \ldots, D_n\}$ of domains and a function *Dom* that maps pairs of the form (property name, type) to elements of D. The sets $D_1, \ldots, D_n$ are called atomic domains (i.e., they contain numeric or symbolic values, but are disjunct from $\mathcal{C}$).
- Γ is a finite set of generic constraints including activation and resource constraints. (Generic constraints are presented in the next section.) ■

Note that both components and their properties are represented by constraint variables. The type of a component is specified by assigning a type symbol from T to that component variable. Depending on its type (assignment), a component will have a varying number of properties, whose existence is guaranteed by satisfying so-called *activation constraints*. The name of a property variable is composed from the name of the originating component and the property name, for example, if property $p \in P$ is defined for component $c \in \mathcal{C}$, then the property variable is named $c.p$. The set of all possible property variables can thus be intensionally specified as $\mathcal{P} = \{c.p \mid c \in \mathcal{C}, p \in P\}$. We adapt the terminology of Mittal and Falkenhainer (1990) and refer to the component and property variables that actually occur in a given constraint network as *active* variables.

Since the existence of property variable $c.p$ depends on the type assigned to c, so does the domain of $c.p$, $dom(c.p)$. It is defined depending on the type of c, that is, $dom(c.p) = Dom(p, \tau)$. Now, $dom(c.p)$ can be either an atomic domain D_i, in which case we call $c.p$ an *attribute* of c, or $dom(c.p) = \mathcal{P}$. In the latter case, we call $c.p$ a *port* of c.

https://doi.org/10.1017/S0890060498124046 Published online by Cambridge University Press

4.1. Generic constraints

Since the exact set of components that exist in a specific solution is not predefined, it is not possible to explicitly write down every constraint that would be involved in the solution. Even if an upper limit were known, we still would have to write down a separate set of constraints for each component constellation. Instead, we specify relationships in terms of *generic constraints*. A generic constraint is a constraint schema where meta-variables act as placeholders for component variables.

DEFINITION 2 (Generic constraint). A generic constraint is a range restricted clause (i.e., a formula of the form $A_1 \wedge \ldots \wedge A_m \rightarrow B_1 \vee \ldots \vee B_n$) where the *A*s are the *negative* literals and the *B*s are the positive ones, containing three kinds of literals:

1. Predicates of the form $\pi(v_1,\ldots,v_k)$, where it can be decided for every assignment $v_1 = x_1,\ldots,v_k = x_k$, whether the assignment satisfies the predicate π. Predicates include equality (for all domains), the *conn* connection predicate, and the arithmetic comparison predicates ($<$, $>$, $\leq$, and $\geq$) for variables with numeric domains. If at least one of the v_i is not active, then $\pi(v_1,\ldots,v_k)$ evaluates as false in every case.
2. The predicate $active(v)$ is true, iff the variable v is active (i.e., selected for solution).
3. $v_1 \equiv v_2$ is true, iff the active variables v_1 and v_2 are identical (note: not only identical assignments). ■

We will show examples of the various possible constraint types below.

DEFINITION 3 (Consistency of generic constraints). Let γ be a range-restricted constraint clause with meta-variables $X_1,\ldots,X_k$. Further, let $V \subseteq \mathscr{C} \cup \mathscr{P}$ be a set of active variables, and $V^C \subseteq V$ is the set of all component variables of V. If t is an assignment tuple for the variables V, a generic constraint γ is said to be satisfied under t, iff

$\forall c_1,\ldots,c_k \in V^C$: $\gamma[X_1|_{c_1},\ldots,X_k|_{c_k}]$ is satisfied under t. ■

In other words, if γ is satisfied for all possible substitutions of the meta-variables with active component variables, then the generic constraint is said to be consistent with respect to the active variables and their assignments. Apart from the fact that the constraint refers to meta-variables, this is the standard DCSP definition of consistency.

We classify GCSP constraints into three groups: activation constraints, compatibility constraints, and resource constraints. The first of these will be examined in more detail, since their relationship to LCON is less obvious than for the other two.

4.2. Activation constraints

Property variables belong to components of specific types. Therefore, if the type of a component is derived (the variable is assigned), the appropriate set of property variables that specify the features and behavior of the component must exist. The existence requirements for the corresponding attributes and ports depending on the type of component are expressed by *activation* constraints in the terminology of standard DCSPs (Mittal & Falkenhainer, 1990).

Let $\mathscr{D} = \langle \mathscr{C}, P, \Gamma \rangle$ be a configuration domain, τ is a type symbol from the domain T of all component variables of $\mathscr{C}$, and p is a property name from P. To state the fact that all components c of type τ have (among others) the property p, the generic constraint for the activation of a property variable $c.p$ has the form

$$C = \tau \overset{\text{RV}}{\Longrightarrow} C.p$$

where C is a meta-variable.[1]

DEFINITION 4 (Consistency of generic activation constraints). Let γ be an activation constraint schema of the form $C = \tau \overset{RV}{\Longrightarrow} C.p$. γ is said to be satisfied, iff for all active component variables c of type τ a property variable $c.p$ is active. ■

We also make the *closed world assumption* in the sense that a solution may only contain property variables which were actually activated by activation constraints.

How do activation constraints translate to the LCON component definitions? Consider the frame example from Section 3.1 (shortened here somewhat to save space).

```
frame F DESCRIPTION:
 F.softwareVersions := {1.0, 2.0, 3.2};
 F.PORTS := {mountedOn[rack],
         slot1[powerSupply], ...
```

This definition corresponds to the set of activation constraints

$$F = Frame \overset{\text{RV}}{\Longrightarrow} F.softwareVersions$$

$$F = Frame \overset{\text{RV}}{\Longrightarrow} F.mountedOn$$

$$F = Frame \overset{\text{RV}}{\Longrightarrow} F.slot1$$

plus the domain definitions of the attribute `softwareVersions` and the various ports:

$$Dom(softwareVersions, Frame) = \{1.0, 2.0, 3.2\}$$

$$Dom(mountedOn, Frame) = \mathscr{P}$$

$$Dom(slot1, Frame) = \mathscr{P}$$

[1] The term "RV" stands for "require-variable." This is the only kind of activation constraint introduced in Mittal and Falkenhainer (1990) that is needed in our approach.

https://doi.org/10.1017/S0890060498124046 Published online by Cambridge University Press

and finally the set of constraints that express the type restrictions on the ports, for example,

$$F = Frame \wedge conn(F, slot1, X, mountedOn) \rightarrow X{=}module$$

$$F = Frame \wedge conn(F, mountedOn, X, Y) \rightarrow X{=}rack$$

4.3. Compatibility constraints

A compatibility constraint is a constraint that does not explicitly influence the existence of variables in the constraint network. In other words, compatibility constraints do not contain the predicate *active*. Compatibility constraints are the workhorses of the representation—they specify what we consider the actual configuration knowledge, namely how components interact with each other.

EXAMPLE. Consider the constraint from Section 3.3.1: "If both `slot15` and `slot16` are occupied, either the same component must be connected to both of them, or it must be of type `moduleA`."

```
frame F DESCRIPTION:
 [ CONN(F,slot15,M15,_)&CONN(F,slot16,M16,_)
   ->(M15 = M16) or is_type(M15,moduleA)];
```

The GCSP form of this constraint would be

$$F = frame \wedge conn(F, F.slot15, M15, _) \wedge conn(F, F.slot16, M16, _)$$
$$\rightarrow (M15 \equiv M16) \vee (M15 = moduleA) \quad \blacksquare$$

The main purpose of this example is to show that the relationship between LCON constraints and a GCSP compatibility constraint is straightforward—there is a one-to-one correspondence between the predicates and operators of the two, with the type that the LCON constraint was defined for now being present as an explicit value in the GCSP constraint.[2]

4.4. Resource constraints

Just as with compatibility constraints, the GCSP form of resource constraints is a direct counterpart of their LCON appearance. Our previous example

```
SUM{M.erlang | M ISA moduleAnalog} >= 5000;
```

corresponds to

$$\left(\sum_{M \in \mathscr{C}: M = moduleAnalog} M.erlang \right) \geq 5000.$$

Resource constraints are a restricted form of set constraints. Their effective handling is made possible by their restriction to one particular domain, the set of component variables $\mathscr{C}$. All resource constraints in a COCOS GCSP problem operate on this one set.

5. REASONING WITH GCSPs

The COCOS inference engine is based on chronological backtracking and constraint propagation through forward checking.

5.1. The COCOS constraint solver

The basic execution cycle in the COCOS system has the same structure as that of any backtracking CSP solver: A variable is selected for assignment, a value is chosen and assigned, and the value is propagated across the constraints that connect that variable to others.

There are three special cases:

1. If the variable was a component variable, assignment to this variable means type assignment. Propagation includes adding property variables appropriate to the new type. Note: this is the only situation when variables are added to the constraint network due to activation constraints.
2. If the variable is a port variable, and a component of the required type is available, it will be connected *via* the port. If no component is available that qualifies for the connection, then a new component will be created and connected. The type of that component may be nonconcrete and will then have to be further restricted later on. For example, if a module exists in an EWSD system and there is no rack in the system that is eligible to contain that module, then a new component will be generated to be connected to the `mountedOn` port of the module. Note that it is this action which guarantees that the requirement in Definition 3 that all variables must be assigned a value is fulfilled. This requirement is completely innocuous in a CSP that is not a GCSP, since there the set of values is fixed. In our case though, the set of components $\mathscr{C}$ is malleable (in fact, the existence of resource constraints allows us to explicitly constrain this set) and the value requirement defines an additional meta-level constraint (i.e., there must be components for the required connections), which is satisfied by the above action.
3. If a resource constraint being checked during propagation can only be satisfied by adding additional components, they will be created (e.g., if the total erlang

[2] Note that unlike Stumptner and Haselböck (1993), we here use the set $\mathscr{P}$ as the domain of port variables, since we want the *conn* constraint (see Section 3.3.1) expresses a direct, bidirectional connection *via* a specific pair of ports. The definition of Stumptner and Haselböck (1993) assigns component variables instead of port variables, so that the assignment is somewhat more flexible (in theory, *n*:1 assignments are possible) at the cost of not specifying the inverse port explicitly. We have since chosen the alternate solution of specifying the port explicitly, so that port variables are assigned the port variable to which they are connected.

https://doi.org/10.1017/S0890060498124046 Published online by Cambridge University Press

values are below 5000 for the resource constraint from Section 3.4, new analog modules will be created).

These three cases provide the connection between the meta-level (reasoning about the contents of the set $\mathcal{C}$) and the component level (reasoning about component properties). Other than that, the same inference mechanism is used to deal with both levels. Note that in two of the special cases, new components are created, and thus variables added to the constraint network. This is the only way in which objects are created apart from the initial network, which may contain initial components (but is not required to). Thus, components are neither fixed as in the common CSP approach, nor are they created through explicit use of side effects. Component creation is not under direct control of the knowledge base designer. If the solver backtracks across a choice that led to creation of a component, that component and its dependent variables are again removed. Note also that the type abstraction mechanism is defined in terms of the constraint network. It thus is also amenable to the same kind of declarative strategy definition—for example, types can be ordered according to the sequence in which their assignment should be attempted.

5.2. Discussing GCSP semantics

It is interesting to contrast the GCSP formalism and the resulting semantics for LCON with other configuration approaches.

1. **Consistency** is defined in terms of satisfying the constraints. Since the variable set is dynamic, it is possible to distinguish between two aspects of consistency, namely a system being

   - *compatible*, that is, its value assignments satisfy the constraints.
   - *stable*, that is, there are no missing variables. Note that while there exist constraints which explicitly influence component generation (see the next point), consistency may also require the creation of components without a constraint being explicitly involved. This is due to the requirement that each variable must be assigned a value, as discussed in the previous section.

2. **Variable generation** involves the different constraint types in different ways. Compatibility constraints do not alter the constraint network. Activity constraints extend the constraint network, but the only variables activated by activity constraints are property variables. Resource constraints explicitly constrain the set of components and can therefore require the creation of components. Finally, the consistency requirement may lead to the creation of new components so a particular port can be assigned a value.

3. **Component properties** (i.e., attributes and ports) are represented by property variables which are activated through use of activation constraints. The GCSP approach therefore distinguishes explicitly between port and component variables depending on their syntactical occurrence in activation constraints (on the left side of these constraints, only component variables occur, and on the right side of these constraints, only property variables occur).

It is clear that the expressiveness of a GCSP is different from that of an ordinary CSP. It should be noted that the approach is also different from the well-known (in particular in the configuration community) "standard" DCSP formalism. First, because the set of variables which is prespecified in a DCSP is not prespecified in a GCSP. There is an additional degree of freedom in the unrestrained number of component variables. Second, the GCSP introduces a semantic distinction between component properties and component identity. This is expressed by the differing syntactic roles of the respective kinds of variables in activation constraints, and by the restriction that only type assignments are used on the left side of activation constraints. This also provides a gain in clarity over the "unstructured" use of activation constraints.

At the same time, while there are aggregate constraints that allow expressing global constraints on the set of components, the focus is different from resource-based systems whose algorithms focus on balancing, since the intent in COCOS was to concentrate on the compositional aspects of the configuration tasks. Components are complex, individual object instances that can be arranged in a complex topology, and handling this kind of knowledge—easily directly expressible in terms of constraints—in COCOS is one of the main issues in configuring technical equipment.

The closest counterpart to the component generation process in COCOS is probably the model generation concept of Klein et al., 1994, which is based on a logical instead of a constraint-based framework. However, the authors point out that the underlying logical framework is quite complex.

5.3. Part-of hierarchies

In Section 2.7, we mentioned the need for a flexible representation of part-of hierarchies. We now have all the knowledge necessary to deal with that problem. A relationship of the kind `X` is-part-of `Y` can be represented by defining a port `mountedOn` for type `X` with domain `Y`, and not allowing this port to remain unconnected. Now, any solution for a constraint network that contains a component `x` of type `X` will need to contain a component of type `Y` connected to `x` *via* `mountedOn`. If none is available, one will be created. Defining another such port called `usedBy` for `X` with domain `Z` results in a second, unrelated part-of relationship, as depicted in Figure 2. Note that these semantics are maintained regardless of whether the superparts or subparts are created first during reasoning.

https://doi.org/10.1017/S0890060498124046 Published online by Cambridge University Press

5.4. Type abstraction and inheritance

We have already referred in Sections 2.6 and 5 to the use of inheritance (a standard feature in object-oriented knowledge representation) in COCOS. A component variable, once created, will be assigned a type. We assume the existence of an inheritance relation $subtype_of(t_1, t_2)$ that defines that t_1 is a direct subtype of type t_2. This relation defines a hierarchy, that is, every type t_1 is a direct subtype of at most one supertype. The leaf types in the hierarchy (i.e., those without subtypes) are called *concrete* types and only these types are assumed to correspond to actual types from the parts catalog. We use $subtype^*$ to refer to the transitive closure of the subtype relation. In general, we refer to t as a subtype of t' if $subtype^*(t, t')$ holds and as a direct subtype of t' if $subtype(t, t')$ holds.

Formally, a not-concrete (also called *intermediate)* type t in the hierarchy possesses the set of properties that corresponds to the intersection of all properties of his subtypes. After a component has been assigned an intermediate type t, its type assignment will later be refined to a subtype of t either by the constraint solver explicitly choosing that variable for refinement, or because filtering removes other subtypes of t from consideration. An alternate approach would be to eliminate all intermediate types and simply enumerate all concrete types as the domain of component variables. Experience seems to indicate that this does not impair performance to a degree that warrants inclusion of the more complex refinement mechanism, but this is subject to further study.

The type mechanism is our main tool for dealing with interchangeability problems. If a new component c_1 is generated, it will subsequently be assigned a type t_1. If this type assignment leads to inconsistency, the system will not generate a further component c_2 (which is again assigned type t_1), but instead will attempt to assign a different type t_2 to c_1 (and, if that fails, the remaining types in the domain of c_1). A different situation occurs when a component that has already been assigned a type t_1 is assigned to a slot, and the assignment attempt fails. This situation is covered in COCOS by an explicit optimization. The failure is analyzed to find out whether it only involves type-specific constraints of the component (instead of individual connections or parameter settings). If this is the case, all components of the same type (t_1) and its subtypes are filtered from the domain of that slot.

The technique of delaying the premature choice of properties that might result in conflicts (this delaying corresponds to the delaying of refining type assignments) is also known as least commitment (Mittal & Frayman, 1987).

6. SYSTEM ARCHITECTURE

Figure 4 gives an overview of the architecture of the system. The heart of the system is the constraint solver which manages the generation and solving of the GCSP constraint network, as described in the previous section. The knowledge base is functionally divided into several parts, namely *domain description*, *requirement specification*, domain-specific *expert heuristics*, and *problem-independent heuristics*. Of these, the first two constitute the model knowledge, which defines the range of possible solutions. The other two specify control knowledge which is aimed at efficiently finding solutions of high quality. We look at the four categories in a bit more detail.

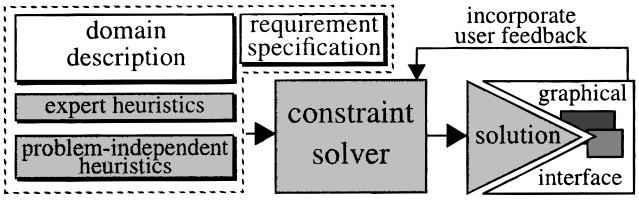


Fig. 4. System architecture.

The domain description is the part of the knowledge base in which the specification of the inventory (i.e., generic knowledge about the component types, their properties and relations) is represented in terms of constraints. On the one hand, this part of the knowledge base contains *catalog knowledge* derived from technical specifications, for example, information about size, ports, and incompatibilities between different types that are based on documented design decisions or physical impossibilities (such as differing requirements on power supply voltage or insufficient space for accommodating a specific subcomponent). On the other hand, it contains expert knowledge based on experience, for example, it may be inadvisable to combine certain component types because heat buildup or electrical interference may occur even though this is not documented in the technical specifications.[3] The requirements for a specific problem instance are also specified in terms of constraints. For example, these may be further restrictions on the component descriptions contained in the domain knowledge, or they may request the existence or nonexistence of component instances.

The search strategy used by the constraint solver can be influenced by providing heuristic control knowledge. Such knowledge will be used when trying to decide between several alternative actions at some point during search. The set of heuristics used in the system can be partitioned into *expert heuristics* and *problem-independent heuristics*. Expert heuristics are those which stem from (human) domain experts and which are highly dependent on the properties and structure of that domain. For instance, sometimes it is efficient to find first a consistent configuration on an abstract (logical or func-

[3] Observing such constraints may be advisable though not absolutely necessary, that is, in some cases they may be ignored when conflicting with other, more basic, requirements. Priorities to resolve such conflicts can be expressed in LCON by using *soft constraints* (also known as *constraint hierarchies)*.

https://doi.org/10.1017/S0890060498124046 Published online by Cambridge University Press

tional) level, and refine that plan afterwards by determining the hardware realization. If the aim is to find only a single solution, because the search space is too large for exhaustive search (as is usually the case), then these kinds of heuristics can also strongly influence the quality of the solution. COCOS allows the specification of value orderings to control search. These orderings can also be intensionally defined (e.g., it is possible to specify that components with minimum cost should be chosen first for a particular port).

Problem-independent heuristics are general techniques developed for the efficient solution of constraint satisfaction problems. Standard heuristics are variable ordering (Dechter & Meiri, 1989) and pruning techniques like forward-checking. In principle, the representation would also be amenable to the use of repair-based constraint algorithms (Minton et al., 1992), although the need so far has not arisen. While the system does not provide an explicit knowledge level representation in the form of the DIDS system (Runkel et al., 1994), the representation is not geared towards a particular solution algorithm.

In Heinrich and Jüngst (1991), a characterization of five different levels of configuration knowledge was proposed. The knowledge contained in configuration expert systems was divided into system knowledge (knowledge about resources, i.e., existence requirements for components), catalog knowledge (i.e., component type descriptions), exception knowledge (expert knowledge about components), evaluation knowledge (which refers to solution quality), and performance knowledge (about effectively finding a configuration). All of these kinds of knowledge are present in our architecture. The constraint formalism is sufficiently powerful to serve as a uniform representation for the first three kinds, while evaluation and performance knowledge correspond to our expert heuristics.

7. EXPERIENCE WITH COCOS

The COCOS tool was implemented in Smalltalk.[4] It provides window-based interfaces for editing the knowledge base, entering configuration requirements, and controlling the configuration process. Knowledge bases, strategies, requirements, and partial or finished configurations can be stored on disk and retrieved. A graphical interface allows fully automatic configuration, checking and completion of manual configurations, or cycles of interactive modification and subsequent checking/completion for finished or partial configurations.

The original COCOS system was used successfully to produce configurations of up to 4000 components, with a running time of several minutes on a SPARCstation 10.

The next generation of the COCOS system uses a redefined version of the LCON language called ConTalk with the original GCSP semantics but the syntax altered to fit that of Smalltalk, simplifying parser maintenance. The tool has now been in production use for two years as part of an application (called LAVA) which consists of the COCOS core system (i.e., the GCSP inference engine) together with application specific administrative tools (i.e., interfaces to other software, report generators, and the LAVA user interface). The LAVA knowledge base covers cabling and additional functional components of the EWSD system. Other knowledge bases that have been developed for COCOS include the configuration of public announcement systems (e.g., loudspeaker systems for airports and railway stations).

In everyday use, LAVA now configures systems containing more than 40,000 components (with several hundred thousand attributes and ports), by partitioning the problem into subproblems called *packages*. Packages are defined according to the major functional groupings of the EWSD system and by thus exploiting the EWSD architecture they can be configured fairly independently. For example, the configurations mentioned in the second paragraph of this section correspond to the core Switching Network (SN) package of a typical EWSD installation. The user interface lets the user decide on the ordering in which packages are to be configured and keeps track of cross-package dependencies that arise during configuration. The package approach is described in more detail in Fleischanderl and Haselböck (1996).

The aim of LAVA was not to replace human configurers (manual configuration being long obsolete in this domain), but the series of table-based configuration systems which were written in C (later C++) and used as input not constraints, but part lists that represented legal partial configurations. A later version of these systems incorporated limited propositional logic reasoning capability. Note that the goal of using an AI approach for this particular domain was not one of superior performance, since the algorithmic systems started out from an advantageous precompiled representation, but maintenance savings in the first place. The old part lists were compiled by hand, which was doubly painful since upgrading to new system versions has to be done by hand as well, and the list representation did not retain the original design criteria (the design constraints having been compiled into the lists), which were lost. Also, various special cases in terms of customer requirements had to be coded into the algorithmic configurators by hand. By comparison, the knowledge base of LAVA incorporates constraints from different departments such as legal constraints from sales and the original technical information (parts catalog and technical constraints) from development, and it is be useful for sales (for a quicker bidding and ordering process), manufacturing (due to the generation of part lists and description of complete assemblies), and service (since it is possible to verify the correctness of existing configurations as well as incorporate strategies that will result in configurations that are easy to maintain).

A feature resulting from the constraint-based architecture is the possibility to manually change a configuration plan and re-start the constraint solver for propagating the

[4]The environment used is Visualworks by ParcPlace (now ObjectShare).

https://doi.org/10.1017/S0890060498124046 Published online by Cambridge University Press

new data and checking consistency. If the changed configuration is not in conformance with the model, the violated constraints are returned. Of course, if a user-manipulated configuration network is consistent but incomplete, it will automatically be completed if wanted. Since virtually all relationships described in the knowledge base are described in terms of constraints, this mechanism could be uniformly implemented for kinds of changes (such as changing component types, restricting attribute values, requesting or unplugging certain connections, etc.).

Altogether, the total development effort of LAVA (even if one includes the application-specific interfaces as well as the development of the generic COCOS tool) was a fraction of that used for the older tools.

The ultimate benefits (development of the configurator in a fraction of the time needed for the older tools, while reaching vastly superior maintainability of the configuration knowledge) were achieved through combining the advantages of an object-oriented representation with clearly defined component structure, grouping of constraints with types, and use of a type hierarchy, with conceptual simplicity of the constraint paradigm. Static and strategic knowledge can be cleanly separated. Certain additional performance gains can be achieved through the use of macro steps - operators which allow a specification of simple-structured partial assemblies that are recommended for duplication *in toto*.

8. CONCLUSION

The aim of the COCOS project was the development of a constraint-based configuration tool for modular technical systems. COCOS uses an object-oriented, constraint-based representation language. The language, LCON, and its new sibling ConTalk, possess semantics based on the generative constraint satisfaction problem (GCSP) formalism. GCSP problems describe configuration domains in term of CSPs which possess a variable set that is split between component variables (which are in principle unlimited in number) and property variables which are used to describe attributes and ports of components. The GCSP approach provides simple semantics for reasoning about configuration problems, while extending traditional CSPs by concepts that allow reasoning about configuration problems with a large and/or variable number of components in their solutions. The GCSP neither corresponds to the standard dynamic CSP (DCSP) nor the resource-based approach, although it effectively subsumes the first and provides features of the second.

At the same time, the underlying formalism is both simple enough and close enough to traditional CSPs to allow direct use or easy adaptation of the existing large body of CSP solving techniques and algorithms. Furthermore, the main advantage of a constraint-based representation, namely the clear separation of problem knowledge and strategic knowledge, is achieved, significantly facilitating knowledge base maintenance and the development of effective strategies for a given configuration problem.

The LAVA configurator based on the COCOS tool has been in production use for two years and provided a drastic reduction both in initial development and knowledge base maintenance effort over earlier tools which used algorithmic or simpler knowledge-based approaches. The EWSD telephone switching systems that are configured by LAVA belong to the largest electronic systems in existence. Apart from the ease of development, LAVA has also proven suitable to supporting all departments involved with the EWSD manufacturing task, by providing a natural representation of the technical constraints and specifications from the design department, verification of the correctness of legacy systems, clear specifications for manufacturing and shipping, up to the automated configuration support in the sales department. At the same time, the potential of the COCOS system is not exhausted as extensions such as optimization issues, the development of algorithms specially tailored for GCSPs, or further improvement of the package technique that was developed to handle extremely large domains, remain as topics for future research.

REFERENCES

Barker, V., & O'Connor, D. (1989). Expert systems for configuration at Digital: XCON and beyond. *Comm. ACM 32(3)*, 298–318.

Birmingham, W.P., Gupta, A.P., & Siewiorek, D.P. (1989). The MICON system for computer design. *IEEE Micro 9(5)*, 61–67.

Bowen, J. (1985). Automated configuration using functional reasoning approach. *Proc. AISB*, pp. 79–106. John Wiley & Sons.

Bowen, J., & Bahler, D. (1991). Conditional existence of variables in generalized constraint networks. *Proc. AAAI*, pp. 215–220.

Brown, D.C., & Chandrasekaran, B. (1989). *Design Problem Solving—Knowledge Structures and Control Strategies*. Wiley.

Cunis, R., Günter, A., Syska, I., Bode, H., & Peters, H. (1989). PLAKON—An approach to domain-independent construction. *Proc. IEA/AIE Conf.*, Tennessee, UTSI.

Dechter, R., & Meiri, I. (1989). Experimental evaluation of preprocessing techniques in constraint satisfaction problems. *Proc. Eleventh IJCAI*, pp. 271–277.

Fleischanderl, G., & Haselböck, A. (1996). Thoughts on partitioning large-scale configuration problems. In Tech Reports FS96-03 AAAI Press, *AAAI Fall Symposium on Configuration*, Boston, Massachusetts.

Frayman, F., & Mittal, S. (1987). COSSACK: A constraint-based expert system for configuration tasks. In *Knowledge-Based Expert Systems in Engineering, Planning, and Design*, (Sriram, D. and Adey, R.A., Eds.), pp. 631–636.

Heinrich, M., & Jüngst, E.W. (1991). A resource-based paradigm for the configuring of technical systems from modular components. *Proc. Seventh IEEE Conf. on AI Applications (CAIA)*, pp. 257–264.

Klein, R., Buchheit, M., & Nutt, W. (1994). Configuration as model construction: The constructive problem solving approach. *Proc. Artificial Intelligence in Design '94*, pp. 201–218.

Marcus, S., Stout, J., & McDermott, J. (1988). VT: An expert elevator designer that uses knowledge-based backtracking. *AI Magazine 9(2)*, 95–111.

McDermott, J. (1982). R1: A rule-based configurer of computer systems. *Artificial Intelligence 19*, 39–88.

Minton, S., Johnston, M.D., Philips, A.B., & Laird, P. (1992). Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence 58*, 161–205.

Mittal, S., & Araya, A. (1986). A knowledge-based framework for design. *Proc. AAAI*, pp. 856–865.

Mittal, S., & Frayman, F. (1987). Making partial choices in constraint reasoning problems. *Proc. AAAI*, pp. 631–636.

Mittal, S., & Frayman, F. (1989). Towards a generic model of configuration tasks. *Proc. Eleventh IJCAI*, pp. 1395–1401. Morgan Kaufmann Publishers, Inc., Los Altos, California.

https://doi.org/10.1017/S0890060498124046 Published online by Cambridge University Press

Mittal, S., & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. *Proc. AAAI Conf.*, pp. 25–32.
Runkel, J.T., Balkany, A., & Birmingham, W.P. (1994). Generating non-brittle configuration-design tools. *Proc. Artificial Intelligence in Design '94*, pp. 183–200.
Schreiber, A.Th., & Birmingham, W.P. (1996). The Sisyphus-VT initiative. *Special Issue, Int. J. Human-Computer Studies*, 44.
Searls, D.B., & Norton, L.M. (1990). Logic-based configuration with a semantic network. *Journal of Logic Programming 8*, 53–73.
Steinberg, L., Langrana, N., & Fisher, G. (1989). *MEET: Decomposition and constraint propagation in mechanical design*. Technical Report WP-TR-121, Rutgers University.
Stumptner, M., & Haselböck, A. (1993). A generative constraint formalism for configuration problems. In *3rd Congress Italian Assoc. for AI*, Volume 729 of *Lecture Notes in AI* (Torasso, P., Ed.), pp. 302–313. Springer-Verlag, Turin, Italy.
Wright, J.R., Weixelbaum, E.S., Brown, K., Vesonder, G.T., Palmer, S.R., Berman, J.I., & Moore, H.G. (1993). A knowledge-based configurator that supports sales, engineering, and manufacturing at AT&T network systems. *Proc. Fifth Conf. on Innovative Applications of AI*. AAAI Press.
Yokoyama, T. (1990). An object-oriented and constraint-based knowledge representation system for design object modeling. *Proc. Sixth CAIA*, pp. 146–152.

Markus Stumptner is an associate professor of computer science at the Vienna University of Technology, where he heads the model-based reasoning group at the Institute for Information Systems. His research interests include configuration, model-based reasoning and software diagnosis, constraint satisfaction, and object-oriented techniques for software and data engineering. He received his MS and PhD in computer science from the Vienna University of Technology. He is a member of the IEEE Computer Society, ACM, and AAAI. Contact him at mst@dbai.tuwien.ac.at.

Gerhard E. Friedrich is a professor of computer science at the University of Klagenfurt, Austria. He manages the Institute for Information Technology and directs the Computer and Manufacturing research group. From 1993 to 1997, he was the head of the technology center for configurators and diagnosis systems at Siemens Austria. His research interests include configuration, planning, and diagnosis. He received his PhD and MS in computer science from the Vienna University of Technology. He is a member of the IEEE Computer Society and the ACM. Contact him at gerhard.friedrich@ifi.uni-klu.ac.at.

Alois Haselböck is a member of the development staff at Siemens Austria. He is mainly interested in knowledge representation and solving techniques for constraint-satisfaction systems. He received his PhD and MS in computer science from the Vienna University of Technology. Contact him at alois.haselboeck@siemens.at.

https://doi.org/10.1017/S0890060498124046 Published online by Cambridge University Press